AN EXTENSIBLE DATA COLLABORATION FRAMEWORK BASED ON SHARED OBJECTS

Marvin Goldin
AMSRD-CER-C2-BC-EXP
Fort Monmouth, New Jersey 07703

Timothy Chase
Dept. of Chemistry, Medical Technology and Physics
Monmouth University
West Long Branch, New Jersey 07733

ABSTRACT

Data sharing and collaboration are vital to the successful implementation of network-centric warfare. This paper discusses a novel approach to data sharing and collaboration based on sharing data *objects* instead of just sharing the data itself. Sharing objects permits not only data sharing, but also the sharing of the methods required to interpret the data. The net result is a shared object framework that enables multiple clients to create data objects locally, drive those objects to an interesting state and then share the objects and their subsequent future state transformations among interested clients. As described, the system is extensible because clients may introduce new objects as needed.

1. INTRODUCTION

The concept of network centric warfare (NCW) relies heavily on shared information. NCW translates information superiority into combat power by effectively linking knowledgeable entities in the battlespace. Sensor netting, data fusion and information management technologies implement the processes for generating better battlespace awareness but it is being able to collaborate this information among individuals that is at the heart of the warfighter's ability to exploit the enhanced awareness (Alberts, 1999).

As data becomes more sophisticated its sharing becomes more difficult. Instead of simply sharing data, applications using the shared data must also have shared code to interpret the data. As an example, consider sharing a high resolution military plan that describes the mission state over the course of the mission. Storage of the time variant plan state data presents a problem. In essence, the problem is how to store the values for the plan state variables as a function of time? In other words, how does the plan data represent a function such as $S = F(t)$ where S is the value of the state variable at time t? Suppose, for example, that the function F represented the current amount of fuel held by a given unit. The value of $F(t)$ would represent the fuel held by the unit at any given time t. How could this be represented in the plan data? There are two ways in which this can be done. The first way is by sampling the data and storing the samples. Nyquist-Shannon sampling theorem says that for bandwidth limited functions sampling at twice the highest component frequency insures that no information will be lost. This means, however, that data consumers require methods to recompose the data back into its original form by interpolating between the sample points. The second approach to storing F is to actually store the method for F itself. This approach requires that clients have access to the method F in order to calculate the required values. The net conclusion in either case is that clients using data require shared methods in order to correctly interpret the data. In short, it is not enough to share just data. In addition to the data, the methods to understand the data must also be shared.

In object-oriented programming, one definition of an object is a collection of data and the set of methods that manipulate and permit access to the data in the object. This realization suggests that an approach to the problem of sharing complex data might lie in a system organized around sharing *objects*. We had felt that this pattern might provide some interesting insights into solutions for the general problem of collaborating data. As a result, we decided to experiment with this architecture by building a shared object framework for the development of typical collaborative tools such as chat and whiteboards. The goal was to try and construct the framework in such a way that it had no coupling to the objects being shared. We believed such an approach would allow the introduction of new objects without changes to the framework.

The Communications-Electronics Research, Development, and Engineering Center (CERDEC) Command and Control Directorate (C2D) is involved in providing surrogate command and control (C2) systems for use in man-in-the-loop (MITL) experiments at various Army

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
01 NOV 2006	**N/A**	**-**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
An Extensible Data Collaboration Framework Based On Shared Objects	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **AMSRD-CER-C2-BC-EXP Fort Monmouth, New Jersey 07703**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM002075., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**UU**	**8**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Battle Lab locations, so we had the unique opportunity to produce a system and then experiment with it in a sophisticated battle simulation.

It has been said that "God made your eyes to plagiarize" so our first steps were to examine successful collaboration systems to see how they addressed the issues associated with sharing sophisticated data. C2D's Combined Arms Planning and Execution system (CAPES) is a good example of such a system and because it was developed in our own organization it was available for close study. CAPES provides an environment in which military planners may collaboratively develop plans without having to be physically near each other. The Command Post of the Future (CPoF) is another example of a richly collaborative application permitting a range of real time information sharing using text, graphics, charts and maps. Architecturally, somewhat more opaque than CAPES, CPoF demonstrates a highly integrated collaborative system.

Both CPoF and CAPES make use of similar architectural patterns to implement collaboration but there are important differences. Figure 1 shows a simplified

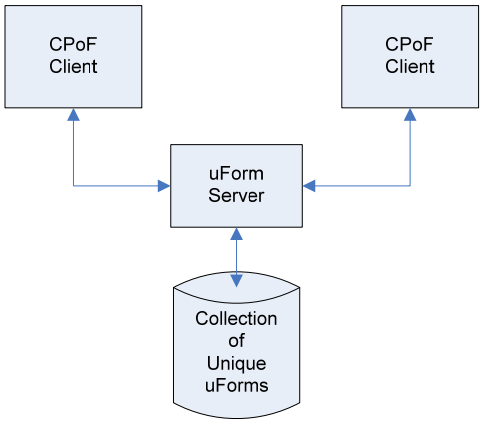

Figure 1. CPoF High-level uForm Sharing

diagram of CPoF's sharing architecture. Sharing is based on a data structure called the *uForm* (Maya). uForms (pronounced *you-form*) are uniquely identified collections of name-value pairs. When a uForm is created it is given an identifier that is unique across all clients. Clients make calls to a framework API to access and change the uForms on the server. Sophisticated client-side buffering makes the operation as efficient as possible. Changes made to the uForm are collaborated through the server to all interested connected clients. CPoF stores virtually everything as uForms including collections of uForms. This architectural regularity permits virtually everything to be collaborated. Unlike objects, however, uForms are purely data and there is no enforced connection between uForm data and methods that manipulate it. A client knows how to process data based on the type of the data. uForms can reference other uForms and this capability is used extensively within CPoF to describe the meaning of data through the concept of *roles*. Code within clients can inspect the uForm roles and make display and processing decisions based on what is found. This capability is similar to the reflection concept found in some programming languages such as Java or C#. Although uForms are not objects because they don't have methods associated with them, the way in which CPoF encodes and shares information exclusively through uForms makes the uForm an interesting pattern.

CAPES takes a similar, but different approach to solving the collaboration problem (C2D, 2002). The high-level CAPES architecture is shown in Figure 2.

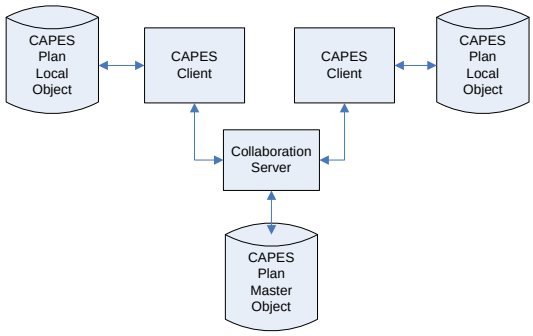

Figure 2. CAPES Collaborative Architecture

As with CPoF, a central server is used but CAPES changes this design by keeping local copies of the plan data on each client. The client application makes changes to the local plan copy, the changes are sent to the master copy and then forwarded to all interested clients. When new clients join a collaborative session they are given a copy of the master plan's current state from the server.

This approach has a number of interesting features. First of all, local copies of the plan may be quickly accessed for read only operations. This is similar to the buffering scheme provided by CPoF, but provides the additional benefit that clients can work in a disconnected mode if communications facilities are lost. Some problems with the CAPES approach include the fact that the only object that is shared is the plan data model (e.g. the approach is not a generalized object-based approach) and the fact that some race conditions will result in the various plan copies becoming unsynchronized. For example, because the client changes its local copy and then collaborates the change to other clients, if client A makes a change at the same time that client B makes a change, A's database will reflect the changes in the order A, B while B's database will reflect the change in order B, A.

2. CATS

Our implementation of the Collaborative Application Tool Set (CATS) attempted to extend good ideas from both CAPES and CPoF while trying to avoid some of those systems' shortcomings. CATS is written in the C# programming language and is based on the .NET 2.0 Common Language Runtime (CLR). This environment was selected because of its rich support for software component management. The .NET CLR provides facilities for object serialization, reflection, custom metadata and custom proxy generation that are powerful, integrated and (sometimes) easy to use. Being a research project, in most cases we opted for implementation expediency rather than real world applicability. This means that it's straightforward to develop CATS-based collaborative applications in C#, but if you want to use a non .NET language for development (can you say Java?) you're currently out of luck.

CATS borrows from CPoF in that almost all functionality is provided in the form of shared objects. Early in CATS' development, however, we discovered that effective collaboration appears to require three fundamental components: shared objects, messaging and centralized file storage. We have not yet tried to provide the messaging and centralized file storage capability through shared objects, but that is a possible future research topic. In what we believe to be an improvement over CPoF's uForms, CATS exchanges serialized objects and remote method invocations rather than just data.

From CAPES, CATS borrows the idea of maintaining local copies of the shared objects with a master copy holding the *official* object state residing on a centralized server. Unlike CAPES, however, changes to shared objects are not performed locally but rather are sent to the server. This means, for example, that when client *C* makes a change to a local shared object the object is not actually changed but instead the change method call is sent to the server. The server sends the change request to all clients who have expressed an interest in the specific object instance. The list of interested clients includes *C* who receives the change notification and performs the required change. The server becomes a sequencing point insuring that each client receives data change requests (method invocations) in the same order as do all other clients. This helps insure that all objects remain synchronized with the server's notion of the object's state.

Our basic design goals for CATS included:

- *Object based.* Shared information would be represented as objects. Client code gains access to object data strictly through object interfaces.

- *Location agnostic objects*. Object code should not care if it is a local non-shared copy, a local copy of a shared object or the master copy of a shared object.

- *Object agnostic framework.* The framework should not have special knowledge about objects that would require its modification when new objects are introduced.

- *Shared/local object reference scheme.* Application code written in the client should reference local objects and shared objects in the same way.

The CATS' high level architecture as shown in Figure 3 meets these goals with various degrees of success.

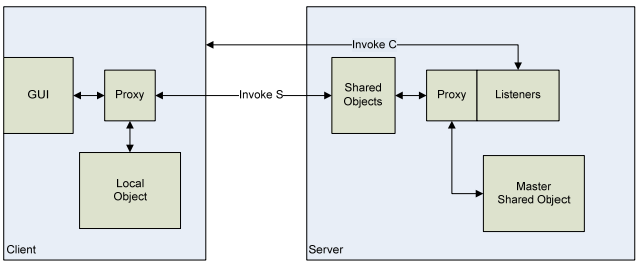

Figure 3. CATS Architecture.

The system architecture is predicated on the following .NET capabilities:

- *Custom proxies*. The ability to generate a proxy that inserts framework code between a client's invocation of a method and the method's actual execution.

- *Object serialization.* The ability to convert an object's state to a form that can be written and read across a communications channel.

- *Reflection*. The ability to be able to inspect an instance of an object and determine compile-time attributes at run time. This includes the ability to discover and call methods.

- *Custom metadata*. The .NET framework provides user-extensible metadata that works with reflection enabling the addition of custom metadata tags to object methods.

2.1 How CATS Works

The basic idea behind CATS is that an instance of a shared object is located on the server. In Figure 3 this is labeled *Master Shared Object*. Similar to CAPES' implementation, the CATS client also maintains a local copy of the shared object. An important CATS' feature is that the object's implementation is the same on either the server or

the client. The same binary code for the object is used on both the client and the server. The local copy of the object is never directly accessed by the client code. Instead the client code makes calls to a proxy. The proxy is generated at runtime by the framework to exactly match the interface implemented by the "real" object. When accessing an object, client code can reference the object's methods and properties to either get data (read reference) or put data (write reference). The proxy examines the client call and if the call is a read, then the proxy directs the call to the appropriate method on the local copy of the object. This makes read references efficient so that GUI-like operations such as screen repaints or scrolling work with local data. The local proxy/instance may be thought of as an object-specific buffering scheme.

If the proxy determines that the object reference is a write operation, then the write is serialized into a message and the message is passed to the server. The message contains the name of the method/property being called and all of the arguments that are being passed to the method. The message is, in effect, a serialization of the method call. When the server receives the message it executes the method call on the master copy of the object causing the desired state change on the master copy. The server then sends the message to every client that has expressed an interest in that particular shared object. The list includes the client that originally initiated the operation. When each client receives the method invocation message, each executes the proper method on the local copy of the object and the operation is complete. Using this technique insures that all of the client's copies of the shared objects are synchronized.

One of the benefits derived from using the proxy scheme is that client code doesn't "know" if it is referencing a shared object via a proxy or a real local object. In fact, the framework creates objects by publishing real local objects. The programmatic steps to create an object are:

1. Create a local object.
2. Make method calls to set its state as desired. These calls all execute locally because the object is not shared.
3. *Publish* the object to the server.

The publish step assigns a unique identifier to the object and serializes the object's initial state to the server. The client code continues using the (now shared) object in the same way as it was using the local object; only now the object is located on the server and locally accessed via a proxy. When a new client accesses the shared object it asks the framework to give it access to the object. The framework locates the object on the server, creates a local version for the accessing client, serializes the current master object's state into the local object, notifies the server that the client is interest in the object and then creates a proxy for the local instance. From this point on the client accesses the shared object as if it were a local object.

Borrowing from CPoF, CATS' framework itself utilizes shared objects to help provide functionality to applications. The layered "call architecture" is illustrated in Figure 4.

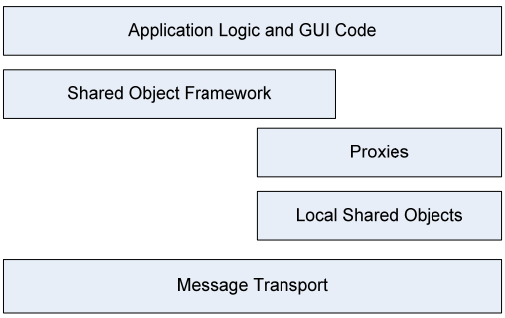

Figure 4. CATS Architectural Layers

Figure 4 shows that the shared object framework actually uses shared objects to provide some of its own functionality. There are several well-known shared objects that contain framework information. For example, the *logged in user list* is an object that contains a list of users logged into the system. Any application needing the list of users can reference this object. The framework itself updates the object whenever a user logs in or out of the system. All of the mechanisms for any shared object are available for the logged in user list. In this way CATS doesn't need to introduce code uniquely designed to maintain its own internal data.

2.2 Chat Example

To better understand how an application uses shared objects we will consider a simple chat program. The chat application we developed to run on CATS is typical and supports features such as lists of users in the chat room, multi-way chatting and "whispering" for private communications. The basic CATS UI with a chat window is shown in Figure 5. The chat application is one of several collaborative tools in CATS. The current CATS UI is broken into two screen panels. On the left, as shown in the figure, is a list of currently visible shared objects. The demonstration version of CATS implements several different types of objects such as chat, whiteboard and shared files. Object names can be hierarchical. For example, the 1stCAB folder in the figure has both fires and intel chat rooms. Users can access the shared objects by dragging them from the left panel to the right panel. In Figure 5, the Example chat room has been dragged into the right side so the user can interact with it.

Figure 5. CATS Is a Multi-document UI

The resulting chat room is pretty typical and provides a list of chat members, an area to enter data and the list of on-going chat messages. The room is implemented as shown in Figure 6.

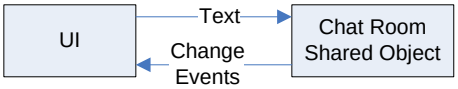

Figure 6. Chat Room Shared Object

The chat room is an instance of a shared chat room object that maintains a list of chat messages and a list of users who are currently logged into the room. The chat UI updates the room object instance by making calls to the room's *add message* method. The UI receives notices of changes to data in the room via a change event. Because the object is shared, any changes made to the room by any client results in all of the clients currently accessing the room being updated. Because the room is really implemented on the server, the server's master copy of the shared object maintains room's state. This provides the unexpected (and useful) feature of being able to see the results of previously on-going chatting immediately upon accessing the room. In other words, when the user accesses a room, the system serializes the room's current state which contains previous chat messages.

All shared objects in CATS are derived from a common base class providing methods for typical object functions such as a method that is called whenever the object is accessed by a client. In the case of chat, the chat room object uses this feature to keep track of the users currently accessing a given chat room.

2.3 Whiteboard Example

Implementation of a whiteboard in CATS demonstrates an additional set of interesting problems. The CATS whiteboard is pretty primitive, but serves as an effective model for a more sophisticated implementation. In many ways a whiteboard is similar to a chat program, except that instead of chat messages being collaborated the whiteboard collaborates the drawable objects appearing on the board.

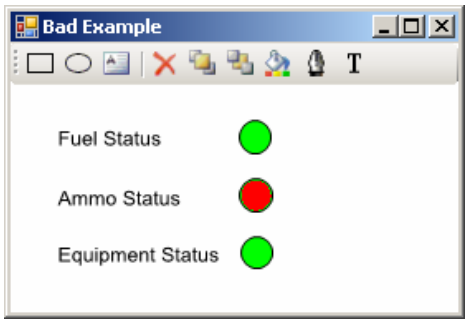

Figure 7. CATS Whiteboard Example

Figure 7 shows a typical CATS whiteboard being used to create a user-defined display. The user has created several indicator shapes (in this case just colored circles). The shapes may be changed to indicate a change in logistical readiness, for example.

Just as with the chat application, the whiteboard application is based on a shared object implementing a container which, in this case, holds the drawables. There are methods to add and remove drawables from the whiteboard and as with the chat example, the UI code calls these methods and, in turn, is called by the instance of the whiteboard object when there are changes made by other clients so that the UI may be updated. This scheme, however, presents an interesting problem. CATS is object based, but there can be no address references to objects. In other words, the collection of objects representing the drawables each must have a unique identifier, but the identifiers cannot be addresses. The red circle in the figure, for example, will have a different address in each client that displays the whiteboard so addresses cannot be used to identify specific instances of objects within containers.

To solve this problem, we introduced the ability to *adorn* methods and properties in shared objects with custom metadata. .NET supports the notion of custom metadata in its custom attribute feature. We defined several custom attributes to give the framework additional information about the intended use of the method or property. In CATS, the custom metadata is largely concerned with where the code for the method is to be executed. We can, for example, specify that a method, even in read mode, is to be executed on the server rather than in the local copy of the object.

This concept was used to generate unique IDs for the drawable objects in the whiteboards. The idea was to generate a guaranteed unique identifier for each drawable. The drawables could then be referenced using that ID and the references would be valid across all clients. A simple method to generate IDs is part of the whiteboard object. The code for the method is shown in Figure 8.

```
[Remote]
[DontCollaborate]
public int GetObjID( ) {
   return ++NextID;
}
private int NextID = 0;
```

Figure 8. Code to Generate IDs

The `[Remote]` adornment indicates to the client proxy that this routine must be executed on the server. Calling the `GetObjID` method from any client via the proxy will result in a message being sent to the server. The `[DontCollaborate]` indicates to the server version of the object that the method invocation should not be collaborated to other clients. The net result is that the `GetObjID` method will be executed on the server for any client call which will result in unique IDs being generated.

The ability to provide the custom metadata tags for methods and properties is an important feature in the CATS framework and helps insure that object code can be the same on both the client and the server. This may be thought of as a way of extending reflection. Because the CATS framework is loosely coupled with the objects it manages, new objects can be added to the application without having to change the underlying framework. The CATS framework "understands" the objects by using reflection to examine the object in order to figure out how the object is to be used. The custom metadata provides a way for the programmer to communicate with the framework.

2.4 Workspaces

Because CATS is object-based we found that we could easily implement collections of shared objects which were themselves shared. This provides a useful workspace concept that effectively mimics CPoF's pasteboard feature. (Maya, CPoF)

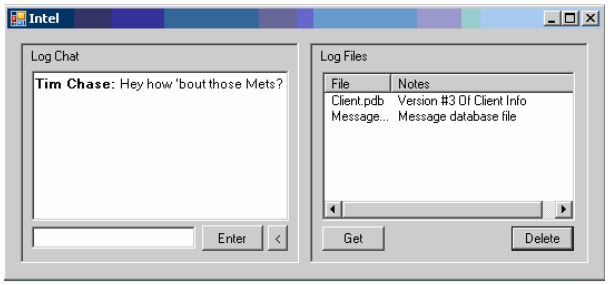

Figure 9. Workspace with Chat and Files

Users can assemble workspaces (collections of shared objects) which are then themselves shared. A good example is a collection of shared files along with a chat room describing their use. In CATS, the workspace is a shared object that is a container for other shared objects. The workspace object contains a list of object identifiers and information about how the objects should be presented in the workspace. This information includes the size of the window displaying the object and the location of the window within the workspace.

Figure 9 shows a workspace with a chat room and a collection of shared files. Shared file collections are another type of CATS shared object that can contain a list of files accessible to any client. The workspace in Figure 9 demonstrates how shared objects may be aggregated together to form higher order shared constructs. One problem that occurs as a consequence of workspace aggregation of shared objects is the behavior when an object is deleted. Currently when objects are deleted all of their references are deleted as well. This means, for example, that when an object in a workspace is deleted, the resulting workspace has a visual "hole" in it where the object used to be. A different approach might be to provide reference counts so that objects aren't deleted until all references to them are also deleted.

3. EXTENSIBILITY

The CATS' capability set is fairly straightforward to augment through the addition of new objects. As it currently works, the CATS application contains two executable files: the server and the client. Both the client and the server are required to have access to .NET assemblies that contain the definition for the objects. .NET's binary object deployment scheme is based on assemblies that behave much like DLLs.

Extensibility is still not as clean as we would like it for two reasons. First are the problems associated with the deployment of new objects. When a programmer develops a new object, the object must be provided to all client locations. A better approach would be to have the object loaded on the server and to then have the code for the object automatically deployed to the clients upon reference. The new MS "One Click Deployment" could be used to implement such a scheme.

The second problem with extensibility comes from how much the system still has to know about the objects. It is true that the framework inspects the objects using reflection and much of the work for integrating objects into the CATS application is done at run time, but there are details that are still not a clean as we would like them. For example, client menus need to be hand modified in order to accommodate new object types. A better approach would be to define interfaces each object implements which de-

scribes the menus and icons that will be used to allow the user to gain access to the objects.

Apart from these two short comings, however, the CATS system is relatively easy to extend. One aspect of extensibility that has proven very useful is the fact each object brings along its own implementation. This means that no client code need be developed to access the shared data that the object represents.

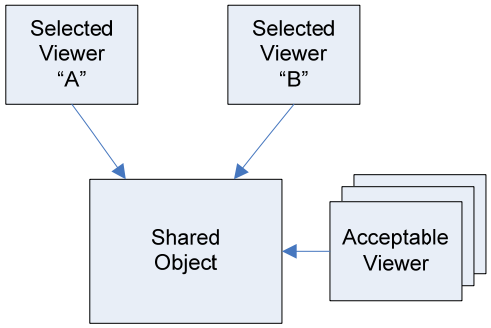

Figure 10. Viewers Display Objects

Although CATS objects simply represent data and as such can be used by any application wishing to access that data, we found it useful to optionally couple *viewers* to the data. One of the methods that each shared object supports is a `GetViewer` method. This method returns a list of viewer objects (.NET controls) that understand how to display the contents of the object. This technique permits users to easily view data objects and is used extensively in the creation and display of workspaces. Dropping an object into a workspace or accessing an object through the CATS UI gives the user access to the appropriate viewer.

Each data object may have any number of selected viewers displaying its state. This means that a single object may support different views. Spot reports, for example, could be viewed geospatially on a map or they could be viewed in a spreadsheet. CATS could support both of these views. In addition, multiple views of the same object are updated at the same time when changes occur to the shared object.

In retrospect, the connection between viewers and data objects may have been made too tightly coupled. A better implementation might have been to have shared objects support multiple interfaces with each interface defining the semantics of that particular type of data. Viewers would be able to display any object implementing a specific interface.

4. ADDITIONS TO SHARED OBJECTS

The CATS system was recently used for a logistical Computer Assisted Map Exercise (CAMEX) at Combat Service Support Battle Lab (CSS-BL). In preparing for the exercise, we developed several collaborative applications based on the CATS framework. These applications included chat, email, and file sharing. Based on this experience, we extended the CATS shared object system to include two features in addition to shared objects. These were:

1. *Central file storage*. Originally, we had thought that we could create shared objects individually representing files, but this quickly became unwieldy. Instead, we opted for a scheme by which files could be moved to the server, assigned a unique id and then be retrieved by any client. This effectively provides a URL-like construct for files that proved useful for file attachments and for file sharing objects.

2. *Messaging*. Although messaging could be implemented using shared objects, we found that given our hub-spoke server pattern, it was more efficient to introduce messaging to the framework. Our messaging system models email in that it provides the ability to send a message object to one or more currently connected users. Messaging is useful for instant message invitations, whispering, and email.

The net result is that the current CATS system provides all three collaborative features: Shared objects, centralized file storage and messaging. Given these basic capabilities, we found implementing collaborative applications to be straightforward. The programming model is easy to understand and being able to create local objects that are then published as shared objects provides the ability to get the object in the proper state before sharing it with others. Arranging objects on a map or whiteboard before sharing is a good example of where this is useful.

5. CONCLUSION

The CATS system has proven useful in an experimental environment and has shown that the shared object paradigm can be used to implement a class of collaborative applications with relative ease. Because each object brings with it code to interpret its data, the introduction of new objects extends the client applications that access them without the need to develop duplicative code. In addition, the objects encapsulate the data thereby making changes to the internal data representation much easier. We could, for example, alter the way in which chat data is stored without altering the client code that uses the data. We could, for example, replace the simple text representations of chat messages with a *rich text* version if we wanted chat to support different fonts and colors.

The pattern implemented by CATS is an interesting one that we will continue to investigate but as is always the case, successes lead to more questions. The CATS imple-

mentation suggests that there is room for additional research in how to more effectively control access to the shared objects and the types of data they contain. This could include addressing the following issues:

- *Dynamically object download*. CATS clients need to have the libraries containing object code present on their computers. An interesting enhancement would be to download object definitions on demand. This would make extensibility even more flexible.

- *Controlled namespaces*. The current CATS system places all objects in a single namespace. It would be interesting to explore how to control access to objects by developing multiple namespaces accessible based on user security credentials.

- *Object ownership*. CATS currently treats all shared objects as common property. This means, for example, that if Marv creates a chat room object, Tim could delete it. Additional work needs to be done to create a sense of ownership for objects and hierarchy of users to control the ownership.

- *Object Locking*. There is no current method to lock objects. As a result, all operations on objects need to be completed in a single method call. A scheme needs to be developed to permit an object to be locked and released. This scheme needs to be robust against the loss of a client currently locking an object.

- *Transactions*. Any sophisticated collaborative application needs some form of transaction processing. This concept is closely aligned with locking. In the CATS context it would mean the ability to treat multiple method calls as a unit that would be deemed successful or not. If not successful, the call would have to be reversed. CATS currently has no facilities for transactional object usage.

- *Live data feeds*. Currently, CATS applications are largely based on humans entering data (chat rooms, whiteboards, etc.) It would be interesting to explore ways in which live data feeds could be published with CATS. We would envision objects that contain various elements of a Common Operating Picture (COP) that are updated by COP reception software and published through changes made to the shared objects.

REFERENCES

Alberts, David S., 1942: Network centric warfare: developing and leveraging information superiority, *CCRP publication series*.

Maya: Visage Design Notes
http://www.maya.com/visage/visage_des/visnote.html

Maya: CPoF Description PDF
http://www.mayaviz.com/web/visualization/downloads/mayaviz_cpof_description.pdf

Command and Control Directorate (C2D): CAPES Users Manual.